# Overview

Diagram

Description automatically generated

* Practice PDD – Pain Driven Development
* Avoid premature optimization

If current design is painful to work with, use principles to guide redesign.

# SRP-Single Responsibility Principle

It simplifies that each software module should have one and only one reason to change.

Two questions around this topic:

1. How to define a C# **module** context?

Does it mean a class, or a function, or something else?

1. What’s the reason to change?

The scenarios we all met before:

* Multipurpose tools don’t perform as well as dedicated tools.
* Dedicated tools are easier to use
* A problem with one part of a multipurpose tool can impact all others

## What’s a Responsibility

It means the purpose of your classes or functions.

* Data Persistence
* Logging
* Validation
* Business logic

Each responsibility could be an axis/reason of change.

## Coupling, Cohesion and Concerns

**Tight coupling**: binds two (or more) details together in a way that’s difficult to change.

**Loose coupling**: offers a modular way to choose which details are involved in a particular operation.

**Separation of Concerns**: Programs should be separated into distinct sections, each addressing a separate concern, or set of information that affects the program.

Cohesive: class elements that belong together

Timeline

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Sample-Class Coupling and Cohesion

Diagram

Description automatically generated

## Class Sample and Problem Analysis

How many responsibilities in RatingEngine.cs?

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

How many responsibilities did you find in RatingEngine. Here are a few that you might have noticed.

1. Everywhere we're doing these Console.WriteLines is an example of how we want to perform logging for the system.

2. The manner in which we're reading data from the file system instead of from perhaps some other source is a persistence responsibility, and if we changed our decision for how we wanted to persist that logic or how we wanted to input policies into this application, we would have to change that implementation detail and touch that code.

3. We've also hardcoded a dependency on the JSON format. If later on we want to use YAML or XML or INI files or some custom binary format, we will again have to change this responsibility.

4. You may have noticed there were a bunch of different business rules encapsulated inside this class as well. Some of these were represented by the different types of policy that were inside of that switch statement. Others were specific formulas that were used to arrive at different ratings.

5. You also, hopefully, saw that there was a number of different kinds of validation going on usually ensuring that the particular properties necessary to perform a rating of a particular type of policy were present inside of that particular case statement inside the switch.

6. And, finally, there was complex logic for things like how to determine someone's age baked into this class. That calculation, although it may not be likely to change, it also is a fairly low-level calculation that maybe doesn't need to be side by side with more high-level concerns of how we want to rate these policies.

## Responsibilities and Testability

Graphical user interface

Description automatically generated with low confidence

Responsibilities have a direct relationship with testability. When classes have many responsibilities, it becomes more difficult to test them. Especially when single methods are doing a lot of different things, it can be very challenging to write tests, in particular unit tests, for them.

Generally, testing classes with many responsibilities results in tests that are longer and more complex. Frequently, the tests are brittle because they are coupled to the implementation, and a change to any responsibility might break tests for any other responsibility that's in that same method or class. This is an example of a test that we might write for the RatingEngine as you've just seen it.

Text

Description automatically generated

Notice that it needs to overwrite the policy.json file on every test run, making running tests in parallel impossible and likely creating problems for us at some point with **locking and contention** that might result in unexpected test failures.

## Solution

### Logging

Public ConsoleLogger Logger {get; set;} = new ConsoleLogger();

Change all the `Console.Writeline()` to `Logger.Log()`

### Persistence

Public class FilePolicySource

{

Public string GetPolicyFromSource()

{

Return File.ReadAllText(“policy.json”);

}

}

Then insert the FilePolicySource in RatingEngine like below:

Public FilePolicySource PolicySource {get; set;} = new FilePolicySource();

String policyJson = PolicySource.GetPolicyFromSource();

I recommend these courses on Refactoring Fundamentals and Microsoft Azure Developer: Refactoring Code.

Summaries

1. Practice Pain Driven Development
2. Each class should have a single responsibility, or reason to change
3. Strive for high cohesion and loose coupling
4. Keep classes small, focused, and testable